TLP **11** (4–5): 713–730, 2011.
doi:10.1017/S1471068411000263
Draft Wed Mar 30 17:21:24 PDT 2011

On the Correctness of Pull-Tabbing

Sergio Antoy
Computer Science Department
Portland State University
Portland, OR 97207, U.S.A.
antoy@cs.pdx.edu

Abstract

Pull-tabbing is an evaluation approach for functional logic computations, based on a graph transformation recently proposed, which avoids making irrevocable non-deterministic choices that would jeopardize the completeness of computations. In contrast to other approaches with this property, it does not require an upfront cloning of a possibly large portion of the choice's context. We formally define the pull-tab transformation, characterize the class of programs for which the transformation is intended, extend the computations in these programs to include the transformation, and prove the correctness of the extended computations.

KEYWORDS: Functional Logic Programming, Non-Determinism, Graph Rewriting, Pull-tabbing

1 Introduction

Functional logic programming (Antoy and Hanus 2010) joins in a single paradigm the features of functional programming with those of logic programming. Logic programming contributes logic variables that are seamlessly integrated in functional computations by narrowing. The usefulness and elegance of programming with narrowing is presented in (Antoy and Hanus 2002; Antoy 2010). At the semantics level free variables are equivalent to *non-deterministic functions* (Antoy and Hanus 2006), i.e., functions that for some arguments may return any one of many results. Thus, at the implementation level variables can be replaced by non-deterministic functions when non-deterministic functions appear simpler, more convenient and/or more efficient to implement (Brassel and Huch 2007). This paper focuses on a graph transformation recently proposed for the implementation of non-determinism of this kind. This transformation is intended to ensure the completeness of computations without cloning too eagerly a large portion of the context of a non-deterministic step. The hope is that steps following the transformation will create conditions that make cloning the not yet cloned portion of the context unnecessary.

2 Motivation

Non-determinism is certainly the most characterizing and appealing feature of functional logic programming. It enables encoding potentially difficult problems into relatively simpler programs. For example, consider the problem of abstracting the dependencies among

the elements of a set such as the functions of a program or the widgets of a graphical user interface. In abstractions of this kind, *component parts* “build” *composite objects*. A non-deterministic function, `builds`, defines which objects are dependent on each part. The syntax is Curry (Hanus 2006).

```
builds p1 = o1
builds p1 = o2
builds p2 = o1
...
```
(1)

A part can build many objects, e.g.: part `p1` builds objects `o1` and `o2`. Likewise, an object can be built from several parts, e.g.: object `o1` is built by parts `p1` and `p2`. Many-to-many relationships, such as that between objects and parts just sketched, are difficult to abstract and to manipulate in deterministic languages. However, in a functional logic setting, the non-deterministic function `builds` is straightforward to define and is sufficient for all other basic functions of the abstraction.

For example, a function that non-deterministically computes a part of an object is simply defined by:

```
is_built_by (builds x) = x
```
(2)

where `is_built_by` is defined using a *functional pattern* (Antoy and Hanus 2005). The set of all the parts of an object is computed by `is_built_by'set`, the implicitly defined *set function* (Antoy and Hanus 2009) of `is_built_by`.

The simplicity of design and ease of coding offered by functional logic languages through non-determinism do not come for free. The burden unloaded from the programmer is placed on the execution. All the alternatives of a non-deterministic choice must be explored to some degree to ensure that no result of a computation goes missing. Doing this efficiently is a subject of active research. Below, we summarize the state of the art.

3 Approaches

There are three main approaches to the execution of non-deterministic steps in a functional logic program. A fourth approach, called *pull-tabbing* (Alqaddoumi et al. 2010), still underdeveloped, is the subject of this paper. Pull-tabbing offers some appealing characteristics missing from the other approaches.

3.1 Running example

We borrow from (Alqaddoumi et al. 2010) a simple example to present the existing approaches and understand their characteristics:

```
flip 0 = 1
flip 1 = 0
coin = 0 ? 1
```
(3)

We want to evaluate the expression

```
(flip x, flip x) where x = coin
```
(4)

We recall that ‘`?`’ is a library function, called *choice*, that returns either of its arguments, i.e., it is defined by the rules:

```
x ? _ = x        rule C1
_ ? y = y        rule C2        (5)
```

and that the `where` clause introduces a *shared* expression. Every occurrence of `x` in (4) has the same value throughout the entire computation according to the *call-time choice* semantics (Hussmann 1992; López-Fraguas et al. 2007). By contrast, in `(flip coin, flip coin)` each occurrence of `coin` is evaluated independently of the other. Fig. 1 highlights the difference between these two expressions when they are depicted as graphs.

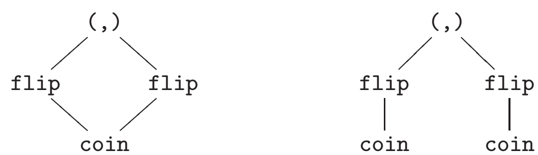

Fig. 1. *Depiction of* (4) (*left*) *and of* `(flip coin, flip coin)` (*right*) *as graphs. The symbol* `(,)` *denotes the pair constructor.*

A *context* is an expression with a distinguished symbol called *hole* denoted '$[\,]$'. If C is a context, $C[x]$ is the expression obtained by replacing the hole in C with x. E.g., the expression in (4) can be written as $C[$`coin`$]$, in which C is `(flip x, flip x) where x =` $[\,]$. The context $[\,]$ is called *empty* context. An expression rooted by a node n labeled by the choice symbol is informally referred to as *a choice* and each argument of the choice symbol, or successor of n, is referred to as a choice's *alternative*.

3.2 Previous approaches

Backtracking is the most traditional approach to non-deterministic computations in functional logic programming. Evaluating a choice in some context, say $C[u?v]$, consists in selecting either alternative of the choice, e.g., u (the criterion for selecting the alternative is not relevant to our discussion), replacing the choice with the selected alternative, which gives $C[u]$, and continuing the computation. In typical interpreters, if and when the computation of $C[u]$ completes, the result is consumed, e.g., printed, and the user is given the option to either terminate the execution or compute $C[v]$. Backtracking is well-understood and relatively simple to implement. It is employed in successful languages such as Prolog (ISO 1995) and in language implementations such as PAKCS (Hanus 2008) and $\mathcal{TOY}$ (Caballero and Sánchez 2007). The major objection to backtracking is its incompleteness. If the computation of $C[u]$ does not terminate, no result of $C[v]$ is ever obtained.

Copying (or *cloning*) is an approach that fixes the inherent incompleteness of backtracking. Evaluating a choice in some context, say $C[u?v]$, consists in evaluating simultaneously (e.g., by interleaving steps) and independently both $C[u]$ and $C[v]$. In typical interpreters, if and when the computation of either completes, the result is consumed, e.g., printed, and the user is given the option to either terminate the execution or continue with the computation of the other. Copying is simpler than backtracking and it is used in some experimental implementations of functional logic languages (Antoy et al. 2005; Tolmach et al. 2004). The major objection to copying is the significant investment of time and memory made when a non-deterministic step is executed. If an alternative of a choice eventually fails,

cloning the context may have been largely useless. For a contrived example, notice that in `1+(2+(...+(`n` ‘div‘ coin)...))` an arbitrarily large context is cloned when the choice is evaluated, but for one alternative this context is almost immediately discarded.

Bubbling is an approach proposed to avoid the drawbacks of backtracking and copying (Antoy et al. 2006; López-Fraguas et al. 2008). Bubbling is similar to copying, in that it clones a portion of the context of a choice to concurrently compute all its alternatives, but the portion of cloned context is typically smaller than the entire context. We recall that in a rooted graph g, a node d is a *dominator* of a node n, *proper* when $d \neq n$, iff every path from the root of g to n contains d. An expression $C[u?v]$ can be seen as $C_1[C_2[u?v]]$ in which the root of C_2 is a dominator of the hole. A trivial case arises when $C_1 = [\,]$ and $C_2 = C$. Evaluating a choice in some context, say $C[u?v]$, distinguishes whether or not C is empty. If C is the empty context, u and v are evaluated simultaneously and independently, as in copying, but there is no context to clone. Otherwise, the evaluation consists in finding C_1 and C_2 such that $C[u?v] = C_1[C_2[u?v]]$ and the root of C_2 is a proper dominator of the choice, and evaluating $C_1[C_2[u]?C_2[v]]$. When C_1 is the empty context, then bubbling is exactly as copying. Otherwise a smaller context, i.e., C_2 instead of C, is cloned. Bubbling intends to reduce cloning in hopes that some alternative of a choice will quickly fail.

An objection to bubbling is the cost of finding a choice's immediate dominator and the risk of paying this cost repeatedly for the same choice. This cost entails traversing a possibly-large portion of the choice's context. Traversing the context is more efficient than cloning it, since cloning requires node construction in addition to the traversal, but it is still unappealing, since the cost of a non-deterministic step is not predictable and it may grow with the size of an expression.

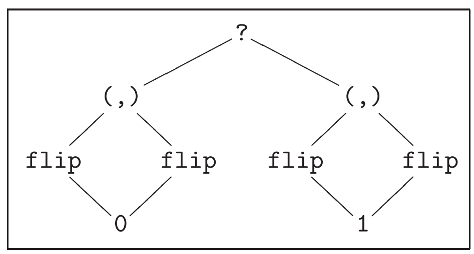

Fig. 2. *Graph depiction of the state of the computation of* (4) *after a bubbling step. Since the dominator of the choice is the root, bubbling and copying are the same for this example.*

3.3 Pull-Tabbing

Pull-tabbing, which is at the core of our work, was first sketched in (Alqaddoumi et al. 2010). The name “pull-tab” originates from the metaphor of pulling the tab of a zipper. For an expression, a choice is a tab and a choice's spine is a zipper. As the tab/choice is pulled up, the zipper/spine opens into two equal strands each of which has a different alternative of the choice at the end.

Evaluating a choice in some context, say $C[u?v]$, distinguishes whether or not C is empty. If C is empty, u and v are evaluated simultaneously and independently, as in copying and bubbling, without any context to clone. Otherwise, the expression to evaluate is of the form $C[s(u?v)]$, for some symbol s (for ease of presentation we assume that s is unary, but there are no restrictions on its arity) and some context C. Pull-tabbing transforms the expression into $C[s(u)?s(v)]$. Without some caution, this transformation is unsound.

Unsoundness may occur when some choice has two predecessors, as in our running example. The choice will be pulled up along two paths creating *two pairs* of strands that eventually must be pair-wise combined together. Some combinations will contain mutually exclusive alternatives, i.e., subexpressions impossible to obtain in languages such as Curry and $\mathcal{TOY}$ that adopt the call-time choice semantics. Fig. 3 presents an example of this situation.

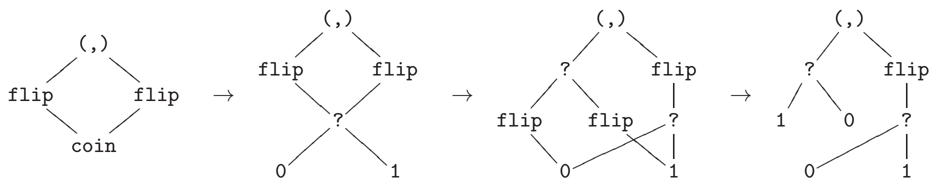

Fig. 3. *Initial portion of the computation of* (4) *with pull-tabbing. The choice in the second expression is being pulled up along the left path to the root. This computation would eventually produce several results including* `(1,0)` *which mixes the left and right alternatives of the same choice.*

We will show that the soundness is recovered if the left and right alternative of a choice are *not* combined in the same expression. To this aim, we attach an identifier to each choice of an expression. We preserve this identifier when a choice is pulled up. If eventually the choice is reduced to either of its alternatives every other choice with the same identifier must be reduced to the same alternative. A very similar idea in a rather different setting is proposed in (Brassel and Huch 2007; Brassel 2011).

A pull-tab step clones a single node, a predecessor of the choice being pulled up. If the choice is pulled all the way up to the root of an expression, the choice's entire spine is cloned. But if an alternative of the choice fails before the choice reaches the root, further cloning of the choice's context becomes unnecessary.

4 Formalization

4.1 Background

We define a term graph in the customary way (Echahed and Janodet 1997), but extend the decorations of nodes with choice identifiers.

Definition 1 (Expression) *Let* Σ *be a* signature*,* $\mathcal{X}$ *a countable set of* variables*,* $\mathcal{N}$ *a countable set of* nodes*,* Ω *a countable set of* choice identifiers*. A* (rooted) graph *over* $\langle\Sigma,\mathcal{N},\mathcal{X},\Omega\rangle$ *is a 5-tuple* $g = \langle\mathcal{N}_g,\mathcal{L}_g,\mathcal{S}_g,\mathcal{R}\mathrm{oots}_g,id_g\rangle$ *such that:*

1. $\mathcal{N}_g \subset \mathcal{N}$ *is the set of nodes of* g*;*
2. $\mathcal{L}_g : \mathcal{N}_g \to \Sigma \cup \mathcal{X}$ *is the* labeling *function mapping each node of* g *to a signature symbol or a variable;*
3. $\mathcal{S}_g : \mathcal{N}_g \to \mathcal{N}_g^*$ *is the* successor *function mapping each node of* g *to a possibly empty string of nodes of* g *such that if* $\mathcal{L}_g(n) = s$*, where* $s \in \Sigma \cup \mathcal{X}$*, and (for the following condition, we assume that a variable has arity zero)* $arity(s) = k$*, then there exist* $n_1,\ldots,n_k$ *in* $\mathcal{N}_g$ *such that* $\mathcal{S}_g(n) = n_1 \ldots n_k$*;*

4. $\mathcal{R}\mathrm{oots}_g \subseteq \mathcal{N}_g$ *is a subset of nodes of* g *called the* roots *of* g*;*
5. $id_g : \mathcal{N}_g \to \Omega$ *is a partial function mapping nodes labeled by the choice symbol to a choice identifier;*
6. *if* $\mathcal{L}_g(n_1) \in \mathcal{X}$ *and* $\mathcal{L}_g(n_2) \in \mathcal{X}$ *and* $\mathcal{L}_g(n_1) = \mathcal{L}_g(n_2)$*, then* $n_1 = n_2$*, i.e., every variable of* g *labels one and only one node of* g*; and*
7. *for each* $n \in \mathcal{N}_g$*, either* $n \in \mathcal{R}\mathrm{oots}_g$ *or there is a path from* r *to* n *where* $r \in \mathcal{R}\mathrm{oots}_g$*, i.e., every node of* g *is reachable from some root of* g*.*

A graph g *is called a* term (graph)*, or more simply an* expression*, if* $\mathcal{R}\mathrm{oots}_g$ *is a singleton.*

Typically we will use "expression" when talking about programs and "graph" when making formal claims. Choice identifiers play a role in computations. Thus, we will explicitly define the id mapping only after formally defining the notion of computation. Term graphs can be seen, e.g., in Figs. 1 and 2. Every choice node of every graph of Fig. 3 would be decorated with the same choice identifier. Choice identifiers are arbitrary and only compared for equality. Node names are arbitrary and irrelevant to most purposes and are typically omitted. However, some definitions and proofs of our claims need to explicitly refer to some nodes of a graph. For this purpose, we adopt the *linear notation for graphs* (Echahed and Janodet 1997, Def. 4). With this convention, the left graph of Fig. 1 is denoted n_0`:(`n_1`:flip` n_2`:coin,` n_3`:flip` n_2`)`, where the node names are the italicized identifiers starting with 'n'. We also make the convention that names of nodes that do not need to be referenced can be omitted, hence `(flip` n_2`:coin, flip` n_2`)`. The latter is conceptually identical to (4). In the linear notation for graphs, infix operators are applied in prefix notation, e.g., see Lemma 3. This practice eases understanding the correspondence between a node identifier and the label of that node.

The definition of graph rewriting (Echahed and Janodet 1997; Plump 1999) is more laborious than, although conceptually very similar to, that of term rewriting (Baader and Nipkow 1998; Bezem et al. 2003; Dershowitz and Jouannaud 1990; Klop 1992). Sections 2 and 3 of (Echahed and Janodet 1997) formalize key concepts of graph rewriting such as *replacement*, *matching*, *homomorphism*, *rewrite rule*, *redex*, and *step* in a form ideal for our discussion. Therefore, we adopt entirely these definitions, including their notations, and only discuss the manipulation of choice identifiers, since they are absent from (Echahed and Janodet 1997).

4.2 Programs

We now formalize the class of rewrite systems that we consider in this paper. A *program* is a rewrite system in a class called *limited overlapping inductively sequential*, abbreviated *LOIS*. In *LOIS* systems, the rules are left-linear and constructor-based (O'Donnell 1985). The left-hand sides of the rules are organized in a hierarchical structure called a *definitional tree* (Antoy 1992) that guides the evaluation strategy (Antoy 2005). In *LOIS* systems, there is a single operation whose rules' left-hand sides overlap. This operation is called *choice*, is denoted by the infix binary operation "?", and is defined by the rules of (5).

LOIS systems have been investigated in some depth. Below we highlight informally the key results that justify our choice of *LOIS* systems.

1. Any *LOIS* system admits a complete, sound and optimal evaluation strategy (Antoy 1997).

2. Any constructor-based conditional rewrite system is semantically equivalent to a *LOIS* system (Antoy 2001).
3. Any *narrowing* computation in a *LOIS* system is semantically equivalent to a *rewriting* computation in another similar *LOIS* system (Antoy and Hanus 2006).

For the above reasons, *LOIS* systems are an ideal core language for functional logic programs. Informally summarizing, *LOIS* systems are general enough to perform any functional logic computation (Antoy 2001) and powerful enough to compute by simple rewriting (Antoy and Hanus 2006) and without wasting steps (Antoy 1997).

4.3 Computations

In our setting, a *computation* of e is a sequence $e = e_0 \Rrightarrow e_1 \Rrightarrow \ldots$ such that $e_i \Rrightarrow e_{i+1}$ is a *step*, i.e., is either one of two graph transformations: a rewrite, denoted by "$\rightarrow$", or a pull-tab, denoted by "$\equiv$". A *rewrite* is the replacement in a graph of an instance of a rewrite rule left-hand side (the *redex*) with the corresponding instance of the rule right-hand side (the *replacement*). The pull-tab transformation is formally defined in the next section. In principle, we do not exclude choice reductions, i.e., non-deterministic steps, but in practice we limit them to the root of an expression. The reason is that reducing a choice makes an irrevocable commitment to one of its alternatives. Pull-tab steps are equivalent to non-deterministic steps in the sense, formally stated and proved in the next section, that they produce all and only the same results, but without any irrevocable commitment.

A computation can be finite or infinite. A computation is *successful* or it *succeeds* iff its last element is a *value*, i.e., a constructor normal form. A computation is a *failure* or it *fails* iff its last element is a normal form with some node labeled by an operation symbol. In non-deterministic programs, such as those considered in this paper, the same expression may have both successful computations and failures. Each expression of a computation is also referred to as a *state* of the computation.

A strategy determines which step(s) of an expression to execute. Essential properties of a strategy, such as to succeed whenever possible, will be recalled in Sec. 5.

4.4 Transformations

As described in Section 4.3, a computation is a sequence of expressions such that each expression of the sequence, except the first one, is obtained from the preceding expression by either of two transformations. One transformation is an ordinary *redex replacement*. We defer to (Echahed and Janodet 1998, Def. 23) the precise formulation of this transformation and to the next section the handling of decorations by this transformation. The second transformation is defined below.

Definition 2 (Pull-tab) *Let g be an expression, n a node of g, referred to as the* target, *not labeled by the choice symbol and $s_1 \ldots s_k$ the successors of n in g. Let i be an index in $\{1, \ldots k\}$ such that s_i, referred to as the* source, *is labeled by the choice symbol and let t_1 and t_2 be the successors of s_i in g. Let g_j, for $j = 1, 2$, be the graph whose root is a fresh node n_j with the same label as n and successors $s_1 \ldots s_{i-1} t_j s_{i+1} \ldots s_k$. Let $g' = g_1 ? g_2$. The* pull-tab *of g with source s_i and target n is $g[n \leftarrow g']$ and we write $g \equiv g[n \leftarrow g']$.*

Fig. 3 depicts the result of a pull-tab step. For a trivial textual example, $((0+2)\,?\,(1+2))*3$ is the pull-tab of $((0\,?\,1)+2)*3$. The definition excludes targets labeled by the choice symbol. These targets are not a problem for the pull-tab transformation, but would complicate, without any benefit, our treatment.

A pull-tab step is conceptually very similar to an ordinary step—in a graph a (sub)graph is replaced. The difference with respect to a rewrite step is that the replacement is not constructed by a rewrite rule, but according to Def. 2. It seems very natural for pull-tab steps too to call *redex* the (sub)graph being replaced.

Term and graph rewriting are similar formalisms that for many problems are able to model the expressions manipulated by functional logic programs. Not surprisingly, expressions are terms in term rewriting and graphs in graph rewriting. A significant difference between these formalisms is the identification of a subexpression of an expression. Term rewriting uses positions, i.e., paths in a tree, whereas graph rewriting uses nodes. Nodes are used for defining both rewrite rules and expressions to evaluate. Nodes are "placed in service" (1) to define rewrite rules, (2) when an expression, called *top-level*, is defined or created for the purpose of a computation, and (3) to define or construct the replacement used in a step. We agree that any node is placed in service *only once*, i.e., the same node is never allocated to distinct top-level expressions and/or replacements. However, the same node may be found in distinct graphs related by a step, since a step makes a localized change in a graph. These stipulations are formalized by the following principle, which is a consequence of placing nodes in service *only once*.

Principle 1 (Persistence) *Let g_1 and g_2 be graphs. If n is a node in $\mathcal{N}_{g_1} \cap \mathcal{N}_{g_2}$, then there exists a graph g such that $g \overset{*}{\Rrightarrow} g_1$ and $g \overset{*}{\Rrightarrow} g_2$.*

4.5 Decorations

To support pull-tabbing and ensure its correctness we attach additional information to an expression. This additional information is formalized as a decoration of a node similar to other decorations present in graph, e.g., label and successors. In this section, we rigorously define the function that maps nodes to choice identifiers.

Definition 3 (Decorations) *Let $A : g_0 \Rrightarrow g_1 \Rrightarrow \ldots$ be a computation. We define the id_{g_i} mapping, for each element g_i of A, by induction on i, as follows: id_{g_i} takes a node of g_i labeled by the choice symbol and produces the node's choice identifier. Base case: $i = 0$. $id_{g_0}(n)$, where n is in g_0 and is labeled by the choice symbol, is an arbitrary element of Ω, provided that id_{g_0} is one-to-one. Ind. case: $i > 0$. By the induction hypothesis, $id_{g_{i-1}}$ is defined for any choice node. In the step $g_{i-1} \Rrightarrow g_i$, whether rewrite or pull-tab, a subexpression of g_{i-1} rooted by a node p is replaced by an expression rooted by a node q. Let n be a node of g_i.*

1. *If n is a node of g_{i-1} labeled by the choice symbol, then $id_{g_i}(n) = id_{g_{i-1}}(n)$.*
2. *Otherwise, if $g_{i-1} \to g_i$ (an ordinary rewrite) and n is labeled by the choice symbol, then $id_{g_i}(n) = \alpha$, for an arbitrary $\alpha \in \Omega$ provided that $id_{g_j}(m) \neq \alpha$ for all $j < i$ and all $m \in g_j$ and $id_{g_i}(n) \neq id_{g_i}(m)$ for $n \neq m$ (i.e., α is fresh).*

3. *Otherwise, if* $g_{i-1} \Rrightarrow g_i$ *(a pull-tab) and* $n = q$*, then* $id_{g_i}(n) = id_{g_{i-1}}(m)$*, where* m *is the source node of the pull-tab.*

The above definition is articulated, but conceptually simple. Below, we give an informal account of it. In a typical step $g \Rrightarrow g'$, most nodes of g end up in g'. The choice identifier, for choices, of these nodes remains the same. In a rewrite, some nodes are created. Any choice node created in the step gets a fresh choice identifier. In a pull-tab, informally speaking, the source (a choice) "moves" and the target (not a choice) "splits." The choice identifier "moves" with its source. Split nodes have no choice identifier.

Each node in the "universe" of nodes $\mathcal{N}$ may belong to several graphs. In (Echahed and Janodet 1997), and accordingly in our extension (see Defs. 1 and 3), the function mapping a node to a decoration depends on each graph to which the node belongs. It turns out that some decorations of a node, e.g., the label, are *immutable*, i.e., the function mapping a node to such decorations does not depend on any graph. We prove the immutability claim for our extension, the choice identifier. Obviously, there is no notion of time when one discusses expressions and considers the decorations of a node. Hence immutable decorations "are set" with the nodes. In practice, these decorations "become known" when a node is "placed in service" for the purpose of a computation or is created by a step.

Lemma 1 (Immutability) *Let* g_1 *and* g_2 *be expressions. If* n *is a node in* $\mathcal{N}_{g_1} \cap \mathcal{N}_{g_2}$*, then* $id_{g_1}(n) = id_{g_2}(n)$.

Proof If a node n belongs to $\mathcal{N}_{g_1} \cap \mathcal{N}_{g_2}$, then, by Principle 1, there exists an expression g and computations $A_1 : g \overset{*}{\Rrightarrow} g_1$ and $A_2 : g \overset{*}{\Rrightarrow} g_2$. By induction on the length of A_1, resp. A_2, using point 1 of Def. 3, $id_{g_1}(n) = id_g(n)$, resp. $id_{g_2}(n) = id_g(n)$. The claim follows by transitivity. □

In view of this result, we drop the subscript from id since this practice simplifies the notation and attests a fundamental invariant.

Pull-tab steps may produce an expression with distinct choices with the same choice identifier. The same identifier tells us that to some extent these redexes are the "same". Therefore, when a computation replaces one such redex with the left, resp. right, alternative, every other "same" redex should be replaced with the left, resp. right, alternative, too. If this does not happen, the computation is unacceptable. The notion of consistency of computations introduced next abstracts this idea.

Definition 4 (Consistency) *A rewrite step that replaces a redex rooted by a node* n *labeled by the choice symbol is called a* choice step. *A computation* A *is* consistent *iff for all* $\alpha \in \Omega$*, there exists an* i (*either 1 or 2*) *such that every choice step of* A *at a node identified by* α *applies rule* C_i *of "*`?`*" defined in* (5).

5 Correctness

A *strategy* determines which step(s) of an expression to execute. A strategy is usually defined as a function that takes an expression e and returns a set S of steps of this expression or, equivalently, the reducts of e according to the steps of S. We will not define any specific

strategy. A major contribution of our work is showing that the correctness of pull-tabbing is strategy-independent.

The classic definition of correctness of a strategy $\mathcal{S}$ is stated as the ability to produce for any expression e (in the domain of the strategy) all and only the results that would be produced by rewriting e. "All and only" leads to the following notions.

Soundness: if $e \overset{*}{\underset{\mathcal{S}}{\rightarrow}} v$ is a computation of e in which each step is according to $\mathcal{S}$ and v is a value (constructor normal form), then $e \overset{*}{\rightarrow} v$.

Completeness: if $e \overset{*}{\rightarrow} v$, where v is a value (constructor normal form), then there exists a computation $e \overset{*}{\underset{\mathcal{S}}{\rightarrow}} v$ in which each step is according to $\mathcal{S}$.

In the definitions of soundness and completeness proposed above, the same expression is evaluated both according to $\mathcal{S}$ and by rewriting. This is adequate with some conventions. Rewriting is not concerned with choice identifiers. This decoration can be simply ignored in rewriting computations. In particular, in rewriting (as opposed to rewriting and pull-tabbing) a computation is always consistent. In graph rewriting, *equality of graphs* is modulo a renaming of nodes. A precise definition of this concept is in (Echahed and Janodet 1997, Sect. 2.5).

Typically, the proof of soundness is trivial for strategies that execute only rewrite steps, but our strategy executes also pull-tab steps, hence it creates expressions that cannot be produced by rewriting. Indeed, some of these expressions will have to be discarded to ensure the soundness. The proof of correctness of pull-tabbing is non-trivial and relies on two additional concepts, *representation* and *invariance*, which are presented in following sections.

5.1 Parallel Moves

Proofs of properties of a computation are often accomplished by "rearranging" the computation's steps in some convenient order. A fundamental result in rewriting, known as the Parallel Moves Lemma (Huet and Lévy 1991), shows that in orthogonal systems the steps of a computation can be rearranged at will. A slightly weaker form of this result carries over to *LOIS* systems. A pictorial representation of this result is provided in Fig. 4. The symbol "$\overset{=}{\rightarrow}$" denotes the reflexive closure of the rewrite relation. The notation "$\overset{=}{\rightarrow}_{n,r}$", where n is a node and r is a rule, denotes either equality or a rewrite step at node n with rule r.

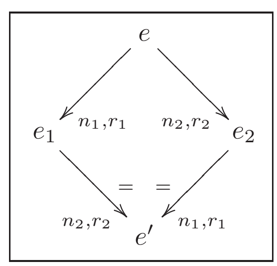

Fig. 4. *The Parallel Moves Lemma for LOIS graph rewriting systems under appropriate conditions on nodes and rules.*

Lemma 2 (*LOIS* parallel moves) *Let e, e_1 and e_2 be expressions such that $e_1 \;{}_{n_1,r_1}\!\!\leftarrow e \rightarrow_{n_2,r_2} e_2$, where for $i = 1, 2$, n_i is a node and r_i is a rule. If $n_1 \neq n_2$ or both $n_1 = n_2$ and $r_1 = r_2$, then there exists an expression e' such that* (*modulo a renaming of nodes*) $e_1 \overset{=}{\rightarrow}_{n_2,r_2} e' \;{}_{n_1,r_1}\!\!\overset{=}{\leftarrow} e_2$.

Proof By cases on the assumption's condition. When both $n_1 = n_2$ and $r_1 = r_2$, the two steps are the same, hence $e_1 = e' = e_2$. When $n_1 \neq n_2$: the claim is a restriction of

(Antoy 1997, Lemma 20) to rewriting in *LOIS* systems. □

5.2 Representation

A characteristic of pull-tabbing, similar to bubbling and copying, is that the completeness of computations is obtained by avoiding or delaying a commitment to either alternative of a choice. In pull-tabbing, similar to bubbling, *both* the alternatives of a choice are kept or "represented" in a *single* expression throughout a good part of a computation. The proof of the correctness of pull-tabbing is obtained by reasoning about this concept, which we formalize below.

Definition 5 (Representation) *We define a mapping R that takes an expression g and returns a set R_g called the* represented set of *g as follows. Let g be an expression. An expression e is in R_g iff there exists a consistent computation $g \overset{*}{\to} e$ (modulo a renaming of nodes) that makes all and only the choice steps of g.*

In other words, we select either alternative for every choice of an expression. For choices with the same identifier, we select the same alternative. Since distinct choice steps occur at distinct nodes, by Lemma 2 the order in which the choice steps are executed to produce any member of the represented set is irrelevant. Therefore, the notion of represented set is *well defined*. The notion of represented set of g is a simple syntactic abstraction not to be confused with the notion of set of values of an expression g (Antoy and Hanus 2009), which is a semantic abstraction fairly more complicated.

5.3 Invariance

The proof of correctness of pull-tabbing is based on two results that informally speaking establish that the notion of represented set is invariant both by pull-tab steps and by non-choice steps.

Lemma 3 (Invariance by pull-tab) *If $g \Rrightarrow g'$ is a pull-tab step, then* (1) *for any expression $e \in R_g$, there exists an expression $e' \in R_{g'}$ such that $e = e'$ (modulo a renaming of nodes), and* (2) *for any expression $e' \in R_{g'}$, there exists an expression $e \in R_g$ such that $e = e'$ (modulo a renaming of nodes).*

Proof We set up the notation for both claims. If $g \Rrightarrow g'$ is a pull-tab step, then by Def. 2:

$$
\begin{aligned}
g &= C[n_f{:}f(s_1,\dots,n{:}?(n_1{:}x,n_2{:}y),\dots s_k)] \\
g' &= C[n'{:}?(n_{f_1}{:}f(s_1,\dots n_1{:}x,\dots s_k),n_{f_2}{:}f(s_1,\dots n_2{:}y,\dots s_k))]
\end{aligned}
$$

where C is some context; n_f, n, n_1, n_2, n', n_{f_1} and n_{f_2} are nodes; $f \neq ?$; $s_1, \dots s_k$ and x and y are expressions; n is the i-th successor of n_f; n_1, resp. n_2, is the i-th successor of n_{f_1}, resp. n_{f_2}.

Claim (1): let $A : g \overset{+}{\to} e$ be a computation witnessing that $e \in R_g$, i.e., a consistent computation making all and only the choice steps of g. From this computation we construct a

computation A' of g' that produces an expression e' satisfying the claim. Without loss of generality, since the notion of represented set is well defined, we assume that the first step of A is $g \rightarrow_{n,C_j} h$, for some h, where C_j is either C_1 or C_2 of (5). Let $\bar{h}$ and h' be expressions defined by the following computation: $g' \rightarrow_{n',C_j} \bar{h} \stackrel{=}{\rightarrow}_{n,C_j} h'$. Node n may or may not be in g'. In particular, n is in g' iff n has more than one predecessor in g. If $n \in \mathcal{N}_{g'}$, then $n \in \mathcal{N}_{\bar{h}}$; otherwise, the step $\bar{h} \stackrel{=}{\rightarrow}_{n,C_j} h'$ does not replace any subexpression of h' and $h' = \bar{h}$. We explicitly construct a homomorphism $\rho : \mathcal{N}_h \rightarrow \mathcal{N}_{h'}$ that shows that $h = h'$ modulo a renaming of nodes. We show that (a) for each node $m \in \mathcal{N}_h$, with $m \neq n_{f_j}$, $m \in \mathcal{N}_{h'}$, and vice versa, (b) for each node $m \in \mathcal{N}_{h'}$, with $m \neq n_f$, $m \in \mathcal{N}_h$. To prove (a), let $m \neq n_f$ be a node of h. Since $g \rightarrow_{n,C_j} h$ is a choice step, m is either in the context C of g or in the subexpression rooted by n_j. These portions of g are preserved by the steps that produce g', $\bar{h}$ and h'. Thus, $m \in \mathcal{N}_{h'}$. The proof of (b) is analogous. Therefore, we define $\rho(n_f) = n_{f_j}$ and $\rho(m) = m$, if $m \neq n_f$. By (a) and (b) and by construction, ρ is a bijection. By construction, ρ preserves root, label, and successors of every node. Thus the computation A' starts with $g \rightarrow \bar{h}$, followed by $\bar{h} \rightarrow h'$ if $\bar{h} \neq h'$. Then, for any step of A starting with expression h at node p with rule r there is a step of A' starting with expression h' at node $\rho(p)$ with rule r. These computations start at equal expressions (modulo a renaming of nodes) and make the same steps, hence they end at equal expressions (modulo a renaming of nodes). Since A is consistent, so is A'. Let e' be the last expression of A'. This proves that $e' \in R_{g'}$.

Claim (2): let $A' : g' \stackrel{+}{\rightarrow} e'$ be a computation witnessing that $e' \in R_{g'}$, i.e., a consistent computation making all and only the choice steps of g'. From this computation we construct a computation A of g that produces an expression e' satisfying the claim. Without loss of generality, since the notion of represented set is well defined, we assume that A' begins with the steps $g' \rightarrow_{n',C_j} \bar{h} \stackrel{=}{\rightarrow}_{n,C_j} h'$. The rule must be the same in both steps because, if $n \in \mathcal{N}_{\bar{h}}$, then $id(n) = id(n')$ and A' is consistent. Node n may or may not be in g'. In particular, n is in g' iff n has more than one predecessor in g. If $n \in \mathcal{N}_{g'}$, then $n \in \mathcal{N}_{\bar{h}}$; otherwise, the step $\bar{h} \stackrel{=}{\rightarrow}_{n,C_j} h'$ does not replace any subexpression of h' and $h' = \bar{h}$. We define the first step of A as $g \rightarrow_{n,C_j} h$. The rest of the proof is substantially equal to that of Claim (1). We complete A with the same steps of A' past h' and obtain an expression e in R_g. We show in exactly the same way that, modulo a renaming of nodes, $h = h'$ and consequently $e = e'$. $\square$

Lemma 4 (Invariance by non-choice) *If $g \rightarrow g'$ is a rewrite non-choice step, then* (1) *for any expression $e \in R_g$, there exists an expression $e' \in R_{g'}$ such that $e \stackrel{*}{\rightarrow} e'$* (*modulo a renaming of nodes*)*, and* (2) *for any expression $e' \in R_{g'}$, there exists an expression $e \in R_g$ such that $e \stackrel{*}{\rightarrow} e'$* (*modulo a renaming of nodes*)*.*

Proof Claim (1): let $A : g = g_0 \rightarrow g_1 \rightarrow \ldots g_n = e$ be a computation witnessing that $e \in R_g$, i.e., a consistent computation making all and only the choice steps of g. From A, we construct a computation A' of g' that produces an expression e' satisfying the claim.

Consider the following diagram, where the top row is A and the bottom row is A':

$$\begin{tikzcd}
g = g_0 \arrow[r] \arrow[d] & g_1 \arrow[r] \arrow[d, "="] & \cdots & g_n = e \arrow[d, "="] & \\
g' = g'_0 \arrow[r, "="] & g'_1 \arrow[r, "="] & \cdots & g'_n \arrow[r, "*"] & e'
\end{tikzcd}$$

By induction on i, for $i = 1, \dots n$, we both define $g'_{i-1} \to g'_i$ and prove that the diagram commutes. To support the induction, we strengthen the statement to include the definition of the step $g_i \overset{=}{\to} g'_i$ and the condition that this step is not a choice step. Base case, $i = 1$: The steps $g'_0 \leftarrow g_0 \to g_1$ are given by the assumptions. Since the first is not a choice step and the second is a choice step, they are at distinct nodes. Hence, Lemma 2 gives the steps $g'_0 \overset{=}{\to} g'_1 \overset{=}{\leftarrow} g_1$ and the commutativity of the diagram. The step $g_1 \overset{=}{\to} g'_1$ is not a choice step because either $g_1 = g'_1$ or it is at the same node as $g_0 \to g'_0$. Ind. case, $i > 1$: The steps $g'_{i-1} \leftarrow g_{i-1} \to g_i$ are given by the induction hypothesis and assumption, respectively. Since one is a choice step and the other is not, they are at distinct nodes. Hence, Lemma 2 gives the steps $g'_{i-1} \overset{=}{\to} g'_i \overset{=}{\leftarrow} g_i$ and the commutativity of the diagram. The step $g_i \overset{=}{\to} g'_i$ is not a choice step because either $g_i = g'_i$or it is at the same node as $g_{i-1} \to g'_{i-1}$. Since A is consistent, A' up to g'_n is consistent as well. We reduce any remaining choice of g'_n consistently with the preceding steps of A', say $g'_n \overset{*}{\to} e'$, to produce an expression $e' \in R_{g'}$. Thus, by the commutativity of the diagram $e = g_n \overset{=}{\to} g'_n \overset{*}{\to} e'$ witnesses the claim.

Claim (2): let $B : g' \overset{*}{\to} e'$ be a computation witnessing that $e' \in R_{g'}$, i.e., a consistent computation making all and only the choice steps of g'. From this computation we construct a computation of g that produces an expression e satisfying the claim. Let $Q = \mathcal{N}_g \cap \mathcal{N}_{g'}$, i.e., be the set of nodes both in g' and g. Suppose that the cardinality of Q is n, for some $n \geqslant 0$. We reorder the steps of B, which is possible by Lemma 2, so that any step at some node of Q occurs before any step at some node not in Q. Let $A' : g' = g'_0 \to g'_1 \to \dots\ g'_n \overset{*}{\to} e'$ be one such computation. From A', we construct a computation A that produces an expression e satisfying the claim. Consider the following diagram, where the top row is A' and the bottom row is A:

$$\begin{tikzcd}
g' = g'_0 \arrow[r] & g'_1 \arrow[r] & \cdots & g'_n \arrow[r, "*"] & e' \\
g = g_0 \arrow[r] \arrow[u] & g_1 \arrow[r] \arrow[u, "="] & \cdots & g_n \arrow[r, "*"] \arrow[u, "="] & e \arrow[ul, "="]
\end{tikzcd}$$

By induction on i, for $i = 1, \dots n$, we both define $g_{i-1} \to g_i$ and prove that the diagram commutes. To support the induction, we strengthen the statement to include the definition of the step $g_i \overset{=}{\to} g'_i$ and the condition that this step is not a choice step. Base case, $i = 1$: Let q_0 be the root of the redex of $g'_0 \to_{q_0,r_0} g'_1$. By assumption, $q_0 \in Q$. Hence $q \in \mathcal{N}_{g_0}$. We let $g_0 \to_{q_0,r_0} g_1$. Thus we have the steps $g'_0 \leftarrow g_0 \to g_1$ where by assumption the first is not a choice step and by construction the second is a choice step. Since these steps are at distinct nodes, by Lemma 2 there exists some g'' such that $g'_0 \overset{=}{\to}_{q_0,r_0} g'' \overset{=}{\leftarrow} g_1$. Therefore, $g'' = g'_1$ and the diagram commutes. The step $g_1 \overset{=}{\to} g'_1$ is not a choice step because either $g_1 = g'_1$ or it is at the same node as $g_0 \to g'_0$. Ind. case, $i > 1$: Let q_{i-1} be the root of the redex of $g'_{i-1} \to_{q_{i-1},r_{i-1}} g'_i$. By assumption, $q_{i-1} \in Q$, hence in g. Thus, node q_{i-1} in g'_{i-1} is not created by the step $g_{i-1} \overset{=}{\to} g'_{i-1}$. Consequently, q_{i-1} is a

node of g_{i-1} too. We let $g_{i-1} \to_{q_{i-1},r_{i-1}} g_1$. Thus we have the steps $g'_{i-1} \leftarrow g_{i-1} \to g_i$ where by assumption the first is not a choice step and by construction the second is a choice step. Since these steps are at distinct nodes, by Lemma 2 there exists some g'' such that $g'_{i-1} \xrightarrow{=}_{q_{i-1},r_{i-1}} g'' \xleftarrow{=} g_i$. Therefore, $g'' = g'_i$ and the diagram commutes. The step $g_i \xrightarrow{=} g'_i$ is not a choice step because either $g_i = g'_i$ or it is at the same node as $g_{i-1} \to g'_{i-1}$. Since A' is consistent, A up to g_n is consistent as well, since corresponding steps use the same rule. We reduce any remaining choice of g_n consistently with the preceding steps of A, say $g_n \xrightarrow{*} e$, to produce an expression $e \in R_g$. We show that $e \xrightarrow{=} g'_n$. If $e \neq g'_n$, then there exists a choice step $g_n \to g_{n+1}$ in A. By construction, this step is at some node q_n which is not in Q and hence is not in g'_n. This means that the step $g_n \to g'_n$ erases node q_n. Thus, we have the steps $g'_n \xleftarrow{=} g_n \to_{q_n,r} g_{n+1}$, for some rule r, where by construction the first is not a choice step and by assumption the second is a choice step. Since these steps are at distinct nodes, by Lemma 2 there exists some g'' such that $g'_n \xrightarrow{=}_{q_n,r} g'' \xleftarrow{=} g_{n+1}$. Since $q_n \notin \mathcal{N}_{g'_n}$, $g'' = g'_n$ and $g_{n+1} \xrightarrow{=} g'_n$. The same above reasoning proves that for any expression h of $g_n \xrightarrow{*} e$ in A, $h \xrightarrow{=} g'_n$. In particular, $e \xrightarrow{=} g'_n$. Thus, $e \xrightarrow{=} g'_n \xrightarrow{*} e'$ witnesses the claim. □

We combine the previous lemmas into computations of any length.

Corollary 1 *If $g \overset{*}{\Rrightarrow} g'$ with no choice steps, then* (1) *for any expression $e \in R_g$, there exists an expression $e' \in R_{g'}$ such that $e \xrightarrow{*} e'$* (*modulo a renaming of nodes*)*, and* (2) *for any expression $e' \in R_{g'}$, there exists an expression $e \in R_g$ such that $e \xrightarrow{*} e'$* (*modulo a renaming of nodes*)*.*

Proof Both claims are proved by a trivial induction on the number of steps of $g \overset{*}{\Rrightarrow} g'$ using Lemmas 3 and 4. □

Theorem 1 (Correctness) *If $g \overset{*}{\Rrightarrow} g'$ with no choice steps, then* (1) *for any value v such that $g \xrightarrow{*} v$ is a consistent computation, there exists a value v' such that $g' \xrightarrow{*} v'$ is a consistent computation, and $v = v'$* (*modulo a renaming of nodes*)*, and* (2) *for any value v' such that $g' \xrightarrow{*} v'$ is a consistent computation, there exists a value v such that $g \xrightarrow{*} v$ is a consistent computation, and $v = v'$* (*modulo a renaming of nodes*)*.*

Proof Claim (1): let $A : g \xrightarrow{*} v$ a consistent computation of g into a value v. By Lemma 2, without loss of generality we assume that $A : g \xrightarrow{*} e \xrightarrow{*} v$, where the segment $g \xrightarrow{*} e$ consists of all the choice steps of g. Since A is consistent, $e \in R_g$. By Corollary 1, there exists a consistent computation $g' \xrightarrow{*} e'$ such that $e = e'$ (modulo a renaming of nodes). Since $e = e'$ (modulo a renaming of nodes) and $e \xrightarrow{*} v$, there exists a computation $e' \xrightarrow{*} v'$ such that $v = v'$ (modulo a renaming of nodes).

Clam (2): the proof is analogous to that of claim (1). □

Theorem 1 suggests to apply both non-choice and pull-tab steps to an expression. Choices pulled up to the root are reduced consistently and without context cloning. Of course, by the time a choice is reduced, all its spines have been cloned—similar to bubbling and copying. A better option, available to pull-tabbing only, is discussed in the next section.

6 Application

The pull-tab transformation is meant to be used in conjunction with some evaluation strategy. We showed that pull-tabbing is not tied to any particular strategy. However, the strategy should be pull-tab-aware in that: (1) a choice should be evaluated (to a head normal form) only when it is *needed* (Antoy 1997), (2) a choice in a root position is reduced (consistently), whereas in a non-root position is pulled, and (3) before pulling a choice, one of the choice's alternatives should be a head-normal form. The formalization of such a strategy would take us well beyond the scope of this paper.

In well-designed, non-deterministic programs, either or both alternatives of most (but not all) choices should fail (Antoy 2010). Under the assumption that a choice is evaluated to a head normal form only when it is *needed* (Antoy 1997), if an alternative of the choice fails, the choice is no longer non-deterministic—the failing alternative cannot produce a value. Thus, the choice can be reduced to the other alternative without loss of completeness and without context cloning. This is where pull-tabbing is advantageous over copying and bubbling—any portion of a choice's context not yet cloned when an alternative fails no longer needs to be cloned. Of course, the implementation must still identify the choice, and choice's single remaining strand as either left or right, to ensure consistency.

7 Related Work

We investigated pull-tabbing, an approach to non-deterministic computations in functional logic programming. Section 3 recalls copying and bubbling, the competitors of pull-tabbing. Here, we briefly highlight the key differences between these approaches. Pull-tabbing ensures the completeness of computations in the sense that no alternative of a choice is left behind until all the results of some other alternative have been produced. Similar to every approach with this property, it must clone portions of the context of a choice. In contrast to copying and bubbling, it clones the context of a choice in minimal increments with the intent and the possibility of stopping cloning the context as soon as an alternative of the choice fails.

The idea of identifying choices to avoid combining in some expression the left and right alternatives of the same choice appears in (Brassel and Huch 2007). The idea is developed in the framework of a natural semantics for the translation of (flat) Curry programs into Haskell. A proof of the correctness of this idea will appear in (Brassel 2011) which also addresses the similarities between the natural semantics and graph rewriting. This discussion, although informal, is enlightening.

8 Conclusion

We formally defined the pull-tab transformation, characterized the class of programs for which the transformation is intended, extended the computations in these programs to include the transformation, proved the correctness of these extended computations, and described the condition that reduces context cloning. In contrast to its competitors, in pull-tabbing any step is a simple and localized graph transformation. This fact should ease executing the steps in parallel. Future work, aims at defining a pull-tab-aware parallel strategy and implementing it to measure the effectiveness of pull-tabbing.

References

ALQADDOUMI, A., ANTOY, S., FISCHER, S., AND RECK, F. 2010. The pull-tab transformation. In *Proceedinsg of the Third International Workshop on Graph Computation Models*. Enschede, The Netherlands, 127–133. Available at `http://gcm-events.org/gcm2010/pages/gcm2010-preproceedings.pdf`.

ANTOY, S. 1992. Definitional trees. In *Proceedings of the Third International Conference on Algebraic and Logic Programming*, H. Kirchner and G. Levi, Eds. Springer LNCS 632, Volterra, Italy, 143–157.

ANTOY, S. 1997. Optimal non-deterministic functional logic computations. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming (ALP'97)*. Springer LNCS 1298, Southampton, UK, 16–30.

ANTOY, S. 2001. Constructor-based conditional narrowing. In *Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01)*. ACM, Florence, Italy, 199–206.

ANTOY, S. 2005. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation 40,* 1, 875–903.

ANTOY, S. 2010. Programming with narrowing. *Journal of Symbolic Computation 45,* 5 (May), 501–522.

ANTOY, S., BROWN, D., AND CHIANG, S. 2006. Lazy context cloning for non-deterministic graph rewriting. In *Proc. of the 3rd International Workshop on Term Graph Rewriting, Termgraph'06*. Vienna, Austria, 61–70.

ANTOY, S. AND HANUS, M. 2002. Functional logic design patterns. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming (FLOPS'02)*. Springer LNCS 2441, Aizu, Japan, 67–87.

ANTOY, S. AND HANUS, M. 2005. Declarative programming with function patterns. In *15th Int'nl Symp. on Logic-based Program Synthesis and Transformation (LOPSTR 2005)*. Springer LNCS 3901, London, UK, 6–22.

ANTOY, S. AND HANUS, M. 2006. Overlapping rules and logic variables in functional logic programs. In *Twenty Second International Conference on Logic Programming*. Springer LNCS 4079, Seattle, WA, 87–101.

ANTOY, S. AND HANUS, M. 2009. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009)*. Lisbon, Portugal, 73–82.

ANTOY, S. AND HANUS, M. 2010. Functional logic programming. *Comm. of the ACM 53,* 4 (April), 74–85.

ANTOY, S., HANUS, M., LIU, J., AND TOLMACH, A. 2005. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*. Springer LNCS 3474, Lubeck, Germany, 108–125.

BAADER, F. AND NIPKOW, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

BEZEM, M., KLOP, J. W., AND DE VRIJER (EDS.), R. 2003. *Term Rewriting Systems*. Cambridge University Press.

BRASSEL, B. 2011. Implementing functional logic programs by translation into purely functional programs. Ph.D. thesis, Brandeis University. (To appear).

BRASSEL, B. AND HUCH, F. 2007. On a tighter integration of functional and logic programming. In *APLAS'07: Proceedings of the 5th Asian conference on Programming languages and systems*. Springer-Verlag, Berlin, Heidelberg, 122–138.

CABALLERO, R. AND SÁNCHEZ, J., Eds. 2007. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*. Available at `http://toy.sourceforge.net`.

DERSHOWITZ, N. AND JOUANNAUD, J. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, J. van Leeuwen, Ed. North Holland, Amsterdam, Chapter 6, 243–320.

ECHAHED, R. AND JANODET, J. C. 1997. On constructor-based graph rewriting systems. Tech. Rep. 985-I, IMAG. Available at ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz.

ECHAHED, R. AND JANODET, J. C. 1998. Admissible graph rewriting and narrowing. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*. MIT Press, Manchester, 325 – 340.

HANUS, M., Ed. 2006. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*. Available at http://www.informatik.uni-kiel.de/~curry.

HANUS, M., Ed. 2008. *PAKCS 1.9.1: The Portland Aachen Kiel Curry System*. Available at http://www.informatik.uni-kiel.de/~pakcs.

HUET, G. AND LÉVY, J.-J. 1991. Computations in orthogonal term rewriting systems. In *Computational logic: essays in honour of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, Cambridge, MA.

HUSSMANN, H. 1992. Nondeterministic algebraic specifications and nonconfluent rewriting. *Journal of Logic Programming 12*, 237–255.

ISO. 1995. Information technology - Programming languages - Prolog - Part 1. General Core. ISO/IEC 13211-1, 1995.

KLOP, J. W. 1992. Term Rewriting Systems. In *Handbook of Logic in Computer Science, Vol. II*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1–112.

LÓPEZ-FRAGUAS, F. J., RODRÍGUEZ-HORTALÁ, J., AND SÁNCHEZ-HERNÁNDEZ, J. 2007. A simple rewrite notion for call-time choice semantics. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM, New York, NY, USA, 197–208.

LÓPEZ-FRAGUAS, F. J., RODRÍGUEZ-HORTALÁ, J., AND SÁNCHEZ-HERNÁNDEZ, J. 2008. Rewriting and call-time choice: The HO case. In *Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*. Springer LNCS 4989, 147–162.

O'DONNELL, M. J. 1985. *Equational Logic as a Programming Language*. MIT Press.

PLUMP, D. 1999. Term graph rewriting. In *Handbook of Graph Grammars*, H.-J. K. H. Ehrig, G. Engels and G. Rozenberg, Eds. Vol. 2. World Scientific, 3–61.

TOLMACH, A., ANTOY, S., AND NITA, M. 2004. Implementing functional logic languages using multiple threads and stores. In *Proc. of the 2004 International Conference on Functional Programming (ICFP)*. ACM, Snowbird, Utah, USA, 90–102.